Constructive Negation in Horn-Clause Programs

Clifford Walinsky

Technical Report No. CS/E 86-003

April, 1986

Technical Report CS/E 86-003 April, 1986

Constructive Negation in Horn-Clause Programs

Clifford Walinsky
Oregon Graduate Center
Beaverton, OR 97006
(503) 690-1121

Abstract A natural mechanism for incorporating negation within Horn-clause programs is presented. Logic programs utilizing constructive negation provide negative as well as positive definitions of predicates. Constructive negation possesses several advantages over the negation mechanism used in standard Horn-clause programs: negation by failure. Important properties of constructive negation are presented. Though negative definitions can be provided by the programmer, a method for inference of negative definitions is described. The inference scheme also precludes production of inconsistent programs.

Constructive Negation

1. Introduction

In this paper I will investigate a sub-class of general logic programs, called *extended Horn-clause* programs. This sub-class is similar to standard Horn-clause programs in many respects. A key difference is the introduction of constructive negation. This mechanism for providing negation is very different from the usual implementation of negation in standard Horn-clause logic [S85]. However, a Horn-clause logic interpreter can execute programs containing constructive negation.

This paper is organized as follows: Section 2 introduces the syntax of extended Horn-clause programs. Sections 3 and 4 describe the declarative and procedural semantics of extended Horn-clause programs. Section 5 demonstrates that the declarative and procedural semantics are in fact equivalent by proving soundness and completeness. Section 6 compares constructive negation with the usual implementation of negation, negation by failure. Extended Horn-clause programs can be inconsistent. To prevent this, inference of a conservative extension is described in section 7. Section 8 describes opportunities for further research. Finally appendix A shows that standard Horn-clause interpreters can execute extended Horn-clause programs.

2. Syntax of Extended Horn-Clause Programs

In this section I will describe the syntactic structure of Horn-clause programs with constructive negation. Such programs will be referred to as *extended* Horn-clause programs. The syntax is very close to standard Horn-clause logic [CM81].

A *term* is either a variable, or a structure of the form $f(t_1,...,t_n)$, where each t_i is a term (when $n = 0$, the term is a constant). An *atom* is of the form $p(t_1,...,t_n)$, where p is a predicate name and each t_i is a term. A negative atom is expressed as ~A, where A is an atom. A *literal* is either an atom or a negative atom. Extended Horn-clause programs are composed of clauses. Each clause is of the form L :- Φ, where L is a literal and Φ is a phrase. If Φ_1 and Φ_2 are phrases, x is a variable, and L is a literal, each phrase is of the form:

Conjunction:	$\Phi_1 \wedge \Phi_2$
Disjunction:	$\Phi_1 \vee \Phi_2$
Existential:	$(\exists x)\ \Phi_1$
Literal:	L

A variable x is *global* within a clause L :- Φ if x occurs within L. A variable x is *bound* within the phrase $(\exists x)\Phi$. A clause L :- Φ is *closed* if every variable in Φ is either global or bound. Henceforth, all program clauses are assumed to be closed.

At this point, the notation of substitutions will be introduced. A *substitution* is a total function mapping expressions to expressions. Every substitution can be denoted by a set of pairs $[e_1/x_1,...,e_n/x_n]$, where each e_i is an expression, and all variables x_i are distinct. Application of a substitution σ to an expression is defined inductively by:

- $\sigma(x) = e$ if σ contains a pair e/x; otherwise, $\sigma(x) = x$.
- $\sigma(f(e_1,...,e_n)) = f(\sigma(e_1),...,\sigma(e_n))$.

Composition of substitutions is defined by: $\sigma \circ \tau(e) = \sigma(\tau(e))$. The *empty* substitution $\varnothing$ is the identity for composition. Expressions e and e' are *unifiable* with substitution σ if $\sigma(e) = \sigma(e')$. A difference operator is also defined: $\sigma - x = [e/y \mid e/y \in \sigma \text{ and } y \neq x]$. Substitutions cannot affect bound variables. If Q is a quantifier, $\sigma((Qx)e) = (Qx)\sigma'(e)$, where $\sigma' = \sigma - x$. A substitution is a *renaming* if the substitution is of the form $[y_1/x_1,...,y_n/x_n]$, where every y_i is a variable.

σ(L :- Φ) is an *instance* of a clause L :- Φ if $\sigma = [t_1/x_1,...,t_n/x_n]$, and each x_i is a global variable. A *ground* expression contains no variables. σ(L :- Φ) is a *ground instance* of a clause L :- Φ if σ(L :- Φ) is a ground clause. σ(L :- Φ) is a *variant* of a clause L :- Φ if σ is a renaming substitution.

3. Declarative Semantics of Extended Horn-Clause Programs

As with general first-order predicate logic, a model-theoretic meaning can be ascribed to each extended Horn-clause program. An *interpretation* is a set of ground literals. For the declarative semantics of extended Horn-clauses alone, every interpretation must contain the distinguished atom *true*. I will denote the fact that a phrase Φ is a valid consequence under interpretation I by $I \models \Phi$. $I \models \Phi$ when $I \models \sigma(\Phi)$ for every ground instance $\sigma(\Phi)$ of Φ. Given that Φ is a ground phrase, Φ is such that $I \models \Phi$ according to the following recursive definition:

- Φ is a literal, and $\Phi \in I$.
- $\Phi = \Phi_1 \bigwedge \Phi_2$, and $I \models \Phi_1$ and $I \models \Phi_2$.
- $\Phi = \Phi_1 \bigvee \Phi_2$, and $I \models \Phi_1$ or $I \models \Phi_2$.
- $\Phi = (\exists x)\Phi_1$, and there is a substitution $\sigma = [t/x]$, with t a ground term, such that $I \models \sigma(\Phi_1)$.

An interpretation I is true of a ground clause L :- Φ if $I \models$ L when $I \models \Phi$. Interpretation I is true of a non-ground clause L :- Φ if I is true of every ground instance of L :- Φ. Finally, interpretation I is true of a program Π if I is true of each clause in Π. An interpretation true of a program Π is a *model* of Π.

For any program Π there is a class of models of Π, $\Sigma(\Pi)$. A distinguished member of this class is the *least model*, $\cap\Sigma(\Pi)$, which is contained in all members of $\Sigma(\Pi)$. Existence and uniqueness of the least model is demonstrated below.

Lemma 3.1: For every program Π, the following is true:

(i) $\cap\Sigma(\Pi)$ is a model.

(ii) $\cap\Sigma(\Pi)$ is the smallest model of Π.

Proof:

(i) Say that $\cap\Sigma(\Pi)$ is not a model. There must be a ground instance L :- Φ of a clause in Π such that $\cap\Sigma(\Pi) \models \Phi$ and $\cap\Sigma(\Pi) \not\models$ L. But for all models M, if M $\models \Phi$, then M $\models$ L. So $\cap\Sigma(\Pi)$

$\models L$, and $\cap\Sigma(\Pi)$ must be a model.

(ii) Say there is a model M' and ground literal L such that $L \notin M'$ and $L \in \cap\Sigma(\Pi)$. There must be a ground instance $L \text{ :- } \Phi$ of a clause in Π such that $\cap\Sigma(\Pi) \models \Phi$. So $M \models \Phi$ for all models M and $M \models L$. Hence M' is not a model. □

The definition of the least model provides a non-effective characterization. An equivalent definition places a structure on the least model. Define the least model to be the smallest interpretation satisfying the inductive criteria below:

$M_0(\Pi) = \{true\}$.

$L \in M_n(\Pi)$ if $L \text{ :- } \Phi$ is a ground instance of a clause in Π and $M_i(\Pi) \models \Phi$ for some $i < n$.

Define $M(\Pi)$ to be the limit of the M_n: There is some finite number N for which $M(\Pi) = M_N(\Pi) = M_{N+k}(\Pi)$ for all non-negative integers k.

In a standard way, I will now show that $M(\Pi)$ is equivalent to $\cap\Sigma(\Pi)$.

Lemma 3.2: For every program Π, $\cap\Sigma(\Pi) = M(\Pi)$.

Proof: The proof is in two parts, first demonstrating $M(\Pi) \subseteq \cap\Sigma(\Pi)$, then demonstrating $\cap\Sigma(\Pi) \subseteq M(\Pi)$.

$M(\Pi) \subseteq \cap\Sigma(\Pi)$: This inclusion is demonstrated by induction on the subsets M_i of $M(\Pi)$.

(Basis) $true \in M_0$, and $true \in M$ for all $M \in \Sigma(\Pi)$.

(Induction) Assume $M_i \subseteq \cap\Sigma(\Pi)$ for all $i < n$. If $L \in M_n$, there is a ground instance $L \text{ :- } \Phi$ of a clause in Π such that $M_i \models \Phi$ for some i. By the hypothesis $M \models \Phi$ for all $M \in \Sigma(\Pi)$. Since every $M \in \Sigma(\Pi)$ is a true interpretation of Π, $L \in M$.

$\cap\Sigma(\Pi) \subseteq M(\Pi)$: I now show that for all phrases Φ such that $M \models \Phi$, there is a minimum integer i such that $M_i \models \Phi$.

(Basis) If $\Phi = true$, then $i = 0$.

(Induction) Assume that for phrases Φ_1 and Φ_2, $M_i \models \Phi_1$ and $M_j \models \Phi_2$. Also assume that $M \models \Phi$ for all $M \in \Sigma(\Pi)$. Now examine each of the phrase forms:

[Φ is a literal L] Since $M \models \Phi$ for all models $M \in \Sigma(\Pi)$, there must be a ground instance $L \text{ :- } \Phi_1$

of a clause in Π such that $M \models \Phi_1$ for all M. Then $M_i \models \Phi_1$ by hypothesis, and $M_{i+1} \models L$.

[$\Phi = \Phi_1 \wedge \Phi_2$] Since $M \models \Phi$ for all models $M \in \Sigma(\Pi)$, $M \models \Phi_1$ and $M \models \Phi_2$. By hypothesis, $M_i \models \Phi_1$ and $M_j \models \Phi_2$. If $i > j$, then $M_i \models \Phi_2$, so $M_i \models \Phi_1 \wedge \Phi_2$.

[$\Phi = \Phi_1 \vee \Phi_2$] Since $M \models \Phi$ for all models $M \in \Sigma(\Pi)$, $M \models \Phi_X$ for $X \in \{1, 2\}$, and all M. By hypothesis $M_k \models \Phi_X$, for $k \in \{i, j\}$. So $M_k \models \Phi_1 \vee \Phi_2$.

[$\Phi = (\exists x)\Phi_1$] Since $M \models \Phi$ for all models $M \in \Sigma(\Pi)$, $M \models \sigma(\Phi_1)$ for some $\sigma = [t/x]$, and all M. By hypothesis, $M_i \models \sigma(\Phi_1)$, so $M_i \models (\exists x)\Phi_1$. □

Let $\Pi \models \Phi$ denote the fact that phrase Φ is a valid consequence of the least model, $M(\Pi)$.

The following lemma will be used in the soundness and completeness proofs to come.

Lemma 3.3: If $L :- \Phi$ is a clause in program Π, and $\Pi \models \tau(\Phi)$ for some substitution τ, then $\Pi \models \tau(L)$.

Proof: For every substitution η such that $\eta \cdot \tau(\Phi)$ is a ground instance, $\Pi \models \eta \cdot \tau(\Phi)$. By definition of $M(\Pi)$, $\Pi \models \eta \cdot \tau(L)$. Since this is true of all substitutions η, $\Pi \models \tau(L)$. □

4. Procedural Semantics of Extended Horn-Clause Programs

Having described the intended connection between logic programs and logic, in this section I will describe the actual execution of logic programs. The procedure for executing a standard Horn-clause logic program is called *SLD-resolution* [EK76], which is a variation of the resolution procedure [R65]. Programs are evaluated with a similar procedure that I will call SLD-resolution with constructive negation (or SLDCN-resolution). SLDCN can be described by an algorithm accepting three arguments: a program, a phrase, and a substitution. When initiated, SLDCN is provided the empty substitution. Successful execution of the SLDCN procedure returns a substitution. When unsuccessful, SLDCN either terminates, returning $\perp$, or does not terminate. I will denote the fact that SLDCN returns substitution τ when given a program Π, a phrase Φ and a substitution σ by $SLDCN(\Pi,\Phi,\sigma) = \tau$ (when program Π is understood, this argument is omitted). The computation procedure of SLDCN is described by the following recursive

rules:

- SLDCN($\Phi, \perp$) = $\perp$.
- SLDCN(*true*, σ) = σ.
- SLDCN($\Phi_1 \wedge \Phi_2$, σ) = SLDCN(Φ_s, SLDCN(Φ_r,σ)), where {r, s} = {1, 2}, and r $\neq$ s.
- SLDCN($\Phi_1 \vee \Phi_2$, σ) = τ if SLDCN(Φ_r, σ) = τ; otherwise SLDCN(Φ_s, σ) if SLDCN(Φ_r, σ) = $\perp$, again where {r, s} = {1, 2}, and r $\neq$ s.
- SLDCN($(\exists x)\Phi$, σ) = SLDCN($\rho(\Phi)$, σ) - y, where $\rho = [y/x]$ and y is a variable not contained in σ or Φ.
- SLDCN(L, σ) = SLDCN(Φ, $\rho \cdot \sigma$) (L is a literal) if there is a variant L′ :- Φ of a clause in Π such that ρ(L′) = $\rho \cdot \sigma$(L) and L′ :- Φ has no variables in common with σ or L; otherwise $\perp$.

  This rule involves use of the ***unification*** algorithm [R65].

The values r and s used in these rules are not necessarily constant. Values of r and s can be any that will lead to successful termination of SLDCN, as if chosen by an oracle. In this paper I will assume the SLDCN procedure relies on such an oracle. By contrast, the Prolog language specifies the values for r and s to be constants 1 and 2, respectively. Consequently, execution of the Prolog interpreter may not terminate in cases when successful execution could be achieved.

To demonstrate this definition of SLDCN, assume that Π contains the clauses:

Example 4.1

(i) `eq(a,a) :- true.`

(ii) `eq(b,b) :- true.`

(iii) `~eq(a,b) :- true.`

(iv) `~eq(b,a) :- true.`

(v) `subsequence(nil,M) :- true.`

(vi)
```
subsequence(cons(X,L), cons(Y,M)) :-
        [eq(X,Y) ∧ subsequence(L,M)]
      ∨ subsequence(cons(X,L),M).
```

(vii) `subsequence(cons(X,L),nil) :- ~true.`

(viii) `~subsequence(cons(X,L),nil) :- true.`

(ix) `~subsequence(nil,M) :- ~true.`

(x)
```
~subsequence(cons(X,L), cons(Y,M)) :-
        [~eq(X,Y) ∨ ~subsequence(L,M)]
      ∧ ~subsequence(cons(X,L),M).
```

This program defines a predicate `subsequence(L,M)`, which is intended to be true when list `L` is a subsequence of list `M`. An empty list is represented by the term `nil`. A nonempty list is represented by the term `cons(`*h*`,`*t*`)` where *h*, the head of the list, is an element from the domain {`a`, `b`}, and *t*, the tail of the list, is also a term representing a list. The following expressions are equivalent, providing a *refutation* of `subsequence(U,cons(b,nil))`

SLDCN(subsequence(U, cons(b, nil)), ∅)

Unification with a variant of clause (vi):
= SLDCN({[eq(X, Y) ∧ subsequence(L′, M′)]
∨ subsequence(cons(X, L′), M′)},
[cons(X, L′)/U, b/Y, nil/M′])

Use of the SLDCN rule for ∨:
= SLDCN([eq(X, Y) ∧ subsequence(L′, M′)],
[cons(X, L′)/U, b/Y, nil/M′])

Use of the SLDCN rule for ∧:
= SLDCN(subsequence(L′, M′), SLDCN(eq(X, Y),
[cons(X, L′)/U, b/Y, nil/M′]))

Unification with a variant of clause (ii):
= SLDCN(subsequence(L′, M′), SLDCN(true,
[cons(b, L′)/U, b/X, b/Y, nil/M′]))

Use of the SLDCN rule for *true*:
= SLDCN(subsequence(L′, M′),
[cons(b, L′)/U, b/X, b/Y, nil/M′])

Unification with a variant of clause (v):
= SLDCN(true,
[cons(b, nil)/U, nil/L′, M′/M", b/X, b/Y, nil/M′])

Use of the SLDCN rule for *true*:
= [cons(b, nil)/U, nil/L′, M′/M", b/X, b/Y, nil/M′]

Consequently, list cons(b, nil) is a subsequence of itself.

5. Soundness and Completeness of SLDCN

In discussing the soundness of the SLDCN procedure, I will need to define the length of a successful refutation. This length is defined to be the number of SLDCN rules invoked during the refutation. For example, the length of the refutation for Example 4.1 is 7. This will now be used to prove the following:

Lemma 5.1: SLDCN($\pi(\Phi)$, σ)$\cdot\tau = \eta$ when SLDCN(Φ, $\sigma\cdot\tau$) $= \eta$.

Proof: By induction on the length of a refutation.

(Basis) Assume the length is 1. Then Φ = *true*, and SLDCN(Φ, $\sigma\cdot\tau$) $= \sigma\cdot\tau$ = SLDCN($\pi(\Phi)$, σ)$\cdot\tau$.

(Induction) Assume the lemma is true for all refutations of length less than n, and a refutation

of phrase Φ under substitution $\sigma \cdot \tau$ is of length n. Now consider each possible phrase form:

[Φ is a literal L] There must be a variant $L' :- \Phi'$ of a clause such that $\rho(L') = \rho \cdot \sigma \cdot \tau(L)$. Then SLDCN(L, $\sigma \cdot \tau$) = SLDCN(Φ', $\rho \cdot \sigma \cdot \tau$). The hypothesis holds, so SLDCN(Φ', $\rho \cdot \sigma \cdot \tau$) = SLDCN($\tau(\Phi')$, $\rho \cdot \sigma) \cdot \tau$. Variables in Φ' are distinct from τ, so SLDCN($\tau(\Phi')$, $\rho \cdot \sigma) \cdot \tau$ = SLDCN(Φ', $\rho \cdot \sigma) \cdot \tau$. Also SLDCN($\tau$(L), $\sigma) \cdot \tau$ = SLDCN(Φ', $\rho \cdot \sigma) \cdot \tau$.

[$\Phi = \Phi_1 \wedge \Phi_2$] Then SLDCN($\Phi$, $\sigma \cdot \tau$) = SLDCN(Φ_s, SLDCN(Φ_r, $\sigma \cdot \tau$)). The hypothesis holds, so SLDCN(Φ_r, $\sigma \cdot \tau$) = SLDCN($\tau(\Phi_r)$, $\sigma) \cdot \tau$, and:

$$\text{SLDCN}(\Phi_s, \text{SLDCN}(\Phi_r, \sigma \cdot \tau)) =$$

$$\text{SLDCN}(\Phi_s, \text{SLDCN}(\tau(\Phi_r), \sigma) \cdot \tau) =$$

$$\text{SLDCN}(\tau(\Phi_s), \text{SLDCN}(\tau(\Phi_r), \sigma)) \cdot \tau =$$

$$\text{SLDCN}(\tau(\Phi_1 \wedge \Phi_2), \sigma) \cdot \tau.$$

[$\Phi = \Phi_1 \vee \Phi_2$] Then SLDCN($\Phi$, $\sigma \cdot \tau$) = SLDCN(Φ_X, $\sigma \cdot \tau$), where $X \in \{1, 2\}$. The hypothesis holds, so SLDCN(Φ_X, $\sigma \cdot \tau$) = SLDCN($\tau(\Phi_X)$, $\sigma) \cdot \tau$ = SLDCN($\tau(\Phi_1 \vee \Phi_2)$, $\sigma) \cdot \tau$.

[$\Phi = (\exists x)\Phi_1$] Then SLDCN(Φ, $\sigma \cdot \tau$) = SLDCN($\rho(\Phi_1)$, $\sigma \cdot \tau)$-y, where $\rho = [y/x]$. The hypothesis holds, so SLDCN($\rho(\Phi_1)$, $\sigma \cdot \tau)$-y = SLDCN($\tau \cdot \rho(\Phi_1)$, $\sigma) \cdot \tau$ - y. The variable y is not contained in τ, so SLDCN($\tau \cdot \rho(\Phi_1)$, $\sigma) \cdot \tau$ - y = SLDCN($\tau((\exists x)\Phi_1)$, $\sigma) \cdot \tau$. □

Several corollaries are now easy to prove and will help in the soundness theorem to come:

Corollary 5.2: SLDCN(Φ, σ) = SLDCN($\sigma(\Phi)$, $\varnothing) \cdot \sigma$.

Proof: Let $\sigma = \sigma \cdot \varnothing$.

Corollary 5.3: If SLDCN(Φ, σ) = τ, then there exists a substitution η such that $\tau = \eta \cdot \sigma$.

Proof: SLDCN(Φ, σ) = SLDCN($\sigma(\Phi)$, $\varnothing) \cdot \sigma$. Let η = SLDCN($\sigma(\Phi)$, $\varnothing$).

I am now ready to provide proof that execution of a logic program provides semantics identical to the intended logical interpretation of the program. This is the soundness property:

Theorem 5.4: SLDCN is sound in that SLDCN(Φ, σ) = τ implies that $\Pi \models \tau(\Phi)$.

Proof: By induction on the length of a refutation.

(Basis) If the length is 1, $\Phi = \mathit{true}$, SLDCN(Φ, σ) = σ, and $\Pi \models \sigma(\mathit{true})$.

(Induction) Assume the theorem is true for all refutations of length less than n, and a refutation of phrase Φ under substitution σ is of length n. Now consider each possible phrase form:

[Φ is a literal L] There must be a variant L′ :- Φ' of a clause such that $\rho(L') = \rho \cdot \sigma(L)$. Then SLDCN(L, σ) = SLDCN(Φ', $\rho \cdot \sigma$) = τ. The hypothesis holds, so $\Pi \models \tau(\Phi')$. By lemma 3.3, $\Pi \models \tau(L')$. I now must show that $\tau(L') = \tau(L)$. By corollary 4.4, there is a substitution η such that $\tau = \eta \cdot \rho \cdot \sigma$. So $\tau(L') = \eta \cdot \rho \cdot \sigma(L')$, and because σ contains no variables occurring in L′, $\eta \cdot \rho \cdot \sigma(L') = \eta \cdot \rho(L')$. Because $\rho(L') = \rho \cdot \sigma(L)$, applying η to both sides gives $\eta \cdot \rho(L') = \eta \cdot \rho \cdot \sigma(L)$. But $\eta \cdot \rho \cdot \sigma = \tau$, so $\tau(L') = \tau(L)$, and $\Pi \models \tau(L)$.

[$\Phi = \Phi_1 \bigwedge \Phi_2$] Then SLDCN($\Phi$, σ) = SLDCN(Φ_s, SLDCN(Φ_r, σ)) = τ. The hypothesis holds, so $\Pi \models \rho(\Phi_r)$ and $\Pi \models \tau(\Phi_s)$, where SLDCN(Φ_r, σ) = ρ. For every substitution η, $\Pi \models \eta \cdot \rho(\Phi_r)$, and by corollary 5.3, at least one substitution η exists such that $\eta \cdot \rho = \tau$. Therefore $\Pi \models \tau(\Phi_r)$, so $\Pi \models \tau(\Phi_1 \bigwedge \Phi_2)$.

[$\Phi = \Phi_1 \bigvee \Phi_2$] Then SLDCN($\Phi$, σ) = SLDCN(Φ_X, σ) = τ, for $X \in \{1, 2\}$. The hypothesis holds, so $\Pi \models \tau(\Phi_X)$, and $\Pi \models \tau(\Phi_1 \bigvee \Phi_2)$.

[$\Phi = (\exists x)\Phi'$] Then SLDCN(Φ, σ) = SLDCN($\rho(\Phi')$, σ) - y = τ, where $\rho = [y/x]$. The hypothesis holds, so $\Pi \models \tau \cdot \rho(\Phi')$. Since y is not contained in τ, $\Pi \models \tau((\exists x)\Phi')$. □

As a result of this theorem, a phrase Φ is ***deducible*** from a program Π by SLDCN if there is a substitution τ such that SLDCN(Φ, $\varnothing$) = τ. τ is also called a ***satisfying*** substitution for Φ.

To provide a completeness proof, I must first demonstrate that SLDCN is monotonic. Define a partial order on substitutions as follows: $\sigma \subseteq \tau$ if and only if there is a substitution η such that $\sigma \cdot \eta = \tau$.

Lemma 5.5: If SLDCN(Φ, σ) = τ and $\sigma' \subseteq \sigma$, then SLDCN(Φ, σ') = τ' for $\tau' \subseteq \tau$.

Proof: By induction on the length of a refutation.

(Basis) If the length is 1, $\Phi = \mathit{true}$. Then SLDCN(true, σ) = σ = τ. And when $\sigma' \subseteq \sigma$,

SLDCN(*true*, σ') = σ' = τ'. So $\tau' \subseteq \tau$.

(Induction) Assume the lemma is true for all refutations of length less than n, and a refutation of phrase Φ under substitution σ is of length n. Now consider each possible phrase form:

[Φ is a literal L] There must be a variant L′ :- Φ' of a clause in Π such that ρ(L′) = $\rho \cdot \sigma$(L) for some substitution ρ. Then SLDCN(L, σ) = SLDCN(Φ', $\rho \cdot \sigma$) = τ. Let $\sigma = \eta \cdot \sigma'$. Then SLDCN(L, σ') = SLDCN(Φ', $\rho \cdot \eta \cdot \sigma'$) = τ, and $\tau \subseteq \tau$.

[$\Phi = \Phi_1 \wedge \Phi_2$] Then SLDCN($\Phi_1 \wedge \Phi_2$, σ) = SLDCN(Φ_s, SLDCN(Φ_r, σ) = τ. Let SLDCN(Φ_r, σ) = ρ. The hypothesis holds, so SLDCN(Φ_r, σ') = ρ', and SLDCN(Φ_s, ρ') = τ', where $\rho' \subseteq \rho$ and $\tau' \subseteq \tau$. Hence, SLDCN($\Phi_1 \wedge \Phi_2$, σ') = τ', and $\tau' \subseteq \tau$.

[$\Phi = \Phi_1 \vee \Phi_2$] Then SLDCN($\Phi_1 \vee \Phi_2$, σ) = SLDCN(Φ_X, σ) = τ, where X $\in$ {1, 2}. The hypothesis holds, so SLDCN(Φ_X, σ') = τ', where $\tau' \subseteq \tau$. SLDCN($\Phi_1 \vee \Phi_2$, σ') = τ', and $\tau' \subseteq \tau$.

[$\Phi = (\exists x)\Phi_1$] Then SLDCN($(\exists x)\Phi_1$, σ) = SLDCN($\rho(\Phi_1)$, σ) - y = τ, where $\rho = [y/x]$. The hypothesis holds, so SLDCN($\rho(\Phi_1)$, σ') - y = τ', where $\tau' \subseteq \tau$. Therefore SLDCN($(\exists x)\Phi_1$, σ') = τ', and $\tau' \subseteq \tau$. □

As a corollary to monotonicity, the following is easily proved:

Corollary 5.6: If SLDCN($\sigma(\Phi)$, $\varnothing$) = τ, then there is a substitution τ' such that SLDCN(Φ, $\varnothing$) = τ' and $\tau' \subseteq \tau \cdot \sigma$.

Proof: By corollary 5.2, SLDCN(Φ, σ) = SLDCN($\sigma(\Phi)$, $\varnothing$)$\cdot \sigma = \tau \cdot \sigma$. And by lemma 5.5, since $\varnothing \subseteq \sigma$, SLDCN($\Phi$, $\varnothing$) = τ', where $\tau' \subseteq \tau \cdot \sigma$. □

Using corollary 5.6 now affords proof of completeness of the SLDCN procedure:

Theorem 5.7: For any ground phrase Φ such that $\Pi \models \Phi$, there is a successful refutation of Φ under substitution $\varnothing$.

Proof: By induction on the structure of the least model M(Π).

(Basis) *true* $\in$ M(Π), and SLDCN(*true*, $\varnothing$) = $\varnothing$.

(Induction) Assume that for all ground phrases Φ_1 and Φ_2 such that $\Pi \models \Phi_1$ and $\Pi \models \Phi_1$, there are successful refutations of Φ_1 and Φ_2 under substitution $\varnothing$. Now consider a phrase Φ for

which $\Pi \models \Phi$, and consider each phrase form:

[Φ is a ground literal L] There must be a ground instance $\rho(L' :- \Phi')$ of a clause in Π such that $\rho(L') = L$ and $\rho(\Phi') = \Phi_1$. Then SLDCN(L, $\varnothing$) = SLDCN(Φ', ρ). By corollary 5.2, SLDCN(Φ', ρ) = SLDCN($\rho(\Phi')$, $\varnothing$)$\cdot\rho$. The hypothesis holds, and SLDCN($\rho(\Phi')$, $\varnothing$)$\cdot\rho = \tau$ = SLDCN(L, $\varnothing$).

[$\Phi = \Phi_1 \bigwedge \Phi_2$] By the hypothesis, SLDCN(Φ_r, $\varnothing$) = σ and SLDCN(Φ_s, $\varnothing$) = τ, where $\{r, s\} = \{1, 2\}$. SLDCN(Φ_s, SLDCN(Φ_r, $\varnothing$)) = SLDCN(Φ_s, σ) = SLDCN($\sigma(\Phi_s)$, $\varnothing$)$\cdot\sigma$, by corollary 5.2. Since, Φ_s is a ground phrase, $\sigma(\Phi_s) = \Phi_s$ and SLDCN($\Phi_1 \bigwedge \Phi_2$, $\varnothing$) = $\tau\cdot\sigma$.

[$\Phi = \Phi_1 \bigvee \Phi_2$] By the hypothesis, SLDCN(Φ_X, $\varnothing$) = τ, where $X \in \{1, 2\}$. Then SLDCN($\Phi_1 \bigvee \Phi_2$, $\varnothing$) = τ.

[$\Phi = (\exists x)\Phi_1$] Let $\Phi_1 = \sigma(\Phi')$. By the hypothesis, SLDCN($\sigma(\Phi')$, $\varnothing$) = τ. Then SLDCN($(\exists x)\Phi'$, $\varnothing$) = SLDCN($\rho(\Phi')$, $\varnothing$) - y, where $\rho = [y/x]$. By corollary 5.6, SLDCN($\rho(\Phi')$, $\varnothing$) - $y = \tau'$, where $\tau' \subseteq \tau\cdot\sigma$. Hence, SLDCN($(\exists x)\Phi'$, $\varnothing$) = τ'. □

6. Negation By Failure

Standard Horn-clause interpreters usually provide negation in a manner different from constructive negation. This section briefly describes their negation and compares it with constructive negation.

In model-theoretic terms, the negated atom $\neg$A is a valid consequence of a program Π, if $\Pi \not\models$ A; and $\Pi \not\models$ A if there is some ground instance A′ of A such that A′ $\notin$ M(Π).

Due to the soundness and completeness of the SLDCN procedure, we could expect similar results for negated atoms: when SLDCN(Π, A, $\varnothing$) returns $\bot$, thus failing to find a satisfying substitution, then $\Pi \models \neg$A. This implementation of negation is referred to as ***negation by failure***. I will denote the fact that a literal L is to be deduced through negation by failure with *not* L. Hence SLDCN(Π, *not* L, σ) returns σ if SLDCN(Π, L, σ) returns $\bot$.

It has been found that soundness and (weak) completeness of negation by failure is obtained for standard SLD-resolution [AE82]. Unfortunately, these promising results do not clearly represent some of the real difficulties involved when using negation. First, there must be

some basis for deciding that *not* L is deducible by negation by failure from program Π if a refutation for L does not terminate. In general this cannot be done. So the soundness and completeness results do not allow for non-termination. It is also not possible in general to determine *a priori* that a given computation will not terminate. A second, and more serious, objection to the use of negation by failure is the new meaning imparted to variables occurring within a query *not* L.

By the soundness of SLDCN-resolution, when literal L is deducible by SLDCN with a satisfying substitution σ, every ground instance of σ(L) is in the least model of program Π. However, no such search is conducted in order to determine if *not* L is deducible. In Example 4.1, deduction of query `not subsequence(L,cons(a,nil))` fails, since SLDCN finds satisfying substitutions `[nil/L]` and `[cons(a,nil)/L]` for query `subsequence(L,cons(a,nil))`. The query `not subsequence(L,cons(a,nil))` would not have failed had `L` been instantiated to `cons(b,nil)`, for example. We see for negation by failure SLDCN-resolution no longer conducts a search for a satisfying substitution.

One possible solution is to employ a *safe* evaluation strategy. This strategy would delay evaluation of any *unsafe* query *not* L until L is a ground literal [C78]. Consequently, certain queries will *flounder*: all possible refutations terminate in unsafe queries. Since it is not generally possible to syntactically detect when a query will flounder, syntactic limitations have been proposed for logic programs that guarantee all queries will not flounder [S85]. I feel these limitations are too severe for general-purpose programming tasks.

Even with a safe evaluation strategy, negation by failure presents severe problems to development of reliable logic programs. To create reliable software and hardware, it is useful to think of distinct components as *black boxes*. This principle can be carried over to Horn-clause programs: the content of each clause may be considered a black box, because we are interested only in the overall behavior of each predicate. The presence of negated atoms in conjunction with negation by failure semantics destroys the ability to consider the content of each clause as

a black box. Whenever a refutation involves negation by failure, the meaning of the variables occurring in the refutation is subtly altered, and the programmer must delve into the contents of clauses to discover where negation by failure arose.

Constructive negation serves as a usable alternative to negation by failure. The procedural semantics of negation are unified with the SLDCN procedure. Hence the "black box" principle is preserved. Further, the notion of floundered queries no longer arises. Several challenges remain for full utilization of constructive negation. First among these challenges is to determine if it is possible to provide negative definitions for large classes of programs. Second, a basis for eliminating logically-inconsistent programs must be established. Fortunately, one idea is sufficient to resolve both problems. This concept, due to David Maier in a private communication, involves extensions of programs.

7. Conservative Extension of Extended Horn-Clause Programs

As a principle of predicate logic, any literal is deducible from an inconsistent program [E72]. A program is inconsistent if both literals A and $\neg$A are deducible. This principle is relevant to extended Horn-clause programs. An interpretation I is *inconsistent* if both $I \models$ A and $I \models$ ~A for some ground atom A. A program Π is *inconsistent* if M(Π) is inconsistent. The following rule for construction of the least model of a program Π duplicates the results of predicate logic for inconsistent programs:

If A,~A $\in$ M(Π), then L $\in$ M(Π) for any ground literal L in the Herbrand base of Π.

The *Herbrand base* of a program Π is the set of all ground literals constructed from predicate and function symbols occurring within Π. Unfortunately, due to issues of undecidability, detection of inconsistent programs cannot generally be performed. But syntactic restrictions on programs eliminate the possibility of constructing syntactically correct inconsistent programs. Of course the problem with this approach is that consistent programs which do not conform to the syntactic criteria will not be accepted.

The *necessary condition* for consistency is as follows:

In a given program, there cannot be a pair of variants A :- Φ and ~A′ :- Φ′ of clauses such that A and A′ are unifiable.

If this condition is observed for a program Π, there is no possibility of having literals A and ~A within the least model of Π. Hence, Π must be consistent.

Unfortunately, the necessary condition is in many cases too strong, eliminating too many programs. In many programs the necessary condition for consistency is violated in the following manner: Given a program with a clause defining a predicate p, $p(x)$:- Φ, there will almost always be a *dual* clause: ~$p(x)$:- Φ′. Frequently there is a definite symmetry between phrases Φ and Φ′. To exploit this symmetry, the dual clauses of a program can be inferred so that consistency of the program is maintained.

I define a model M′ to be an *extension* of a model M if $M \subseteq M'$. Similarly, a program Π′ is an extension of a program Π if $M(\Pi) \subseteq M(\Pi')$. An extension Π′ of a program Π is *conservative* if Π′ is consistent whenever Π is consistent.

For notational convenience, define the total function NOT with one argument, a literal, returning a literal, according to the following rules:

- NOT(A) = ~A.
- NOT(~A) = A.

where A is an atom.

Also, define a binary relation NEG on phrases. Denote by NEG(Φ) = Φ′ the fact that $(\Phi,\Phi') \in$ NEG. NEG is the largest relation for which the *negation condition* holds for all tuples in NEG:

For all consistent interpretations I and phrases Φ, $I \models \Phi$ implies $I \not\models$ NEG(Φ).

Due to the maximality of NEG, if I is consistent, and $I \models \Phi$ implies $I \not\models \Phi'$, then NEG(Φ) = Φ′, and NEG(Φ′) = Φ.

To characterize the negation relation NEG, every tuple in NEG must be an instance of one of the following rules:

(i) $\text{NEG(L)} = \text{NOT(L)}$.

(ii) $\text{NEG}(\Phi_1 \wedge \Phi_2) = \text{NEG}(\Phi_1) \vee \text{NEG}(\Phi_2)$.

(iii) $\text{NEG}(\Phi_1 \vee \Phi_2) = \text{NEG}(\Phi_1) \wedge \text{NEG}(\Phi_2)$.

(iv) $\text{NEG}((\exists x)(\Phi_1 \wedge \Phi_2)) = (\forall x)[\Phi_1 \rightarrow \text{NEG}(\Phi_2)]$.

(v) $\text{NEG}((\exists x)(\Phi_1 \wedge \Phi_2)) = (\forall x)[\Phi_2 \rightarrow \text{NEG}(\Phi_1)]$.

(vi) $\text{NEG}((\forall x)(\Phi_1 \rightarrow \Phi_2)) = (\exists x)[\Phi_1 \wedge \text{NEG}(\Phi_2)]$.

Note that these rules introduce universal quantification. A brief presentation of the declarative semantics of universal quantification is included here. Assume $(\forall x)(\Phi_1 \rightarrow \Phi_2)$ is a ground phrase, and I is an interpretation.

Then $I \models (\forall x)(\Phi_1 \rightarrow \Phi_2)$ if and only if $I \models \sigma(\Phi_2)$ whenever $I \models \sigma(\Phi_1)$ for all substitutions $\sigma = [t/x]$.

With this brief description of universal quantification, I now provide proof that every tuple in NEG satisfies the negation condition.

Lemma 7.1: If $\text{NEG}(\Phi) = \Phi'$ is an instance of a rule for the negation relation, then $I \models \Phi$ implies $I \not\models \Phi'$ for all consistent interpretations I.

Proof: The proof proceeds by induction on the structure of the NEG relation.

(Basis) Consider a literal L. Assume $I \models$ L. Since I is consistent, $I \not\models \text{NOT(L)}$, so according to rule (i), $I \not\models \text{NEG(L)}$.

(Induction) Assume the negation condition is maintained for phrases Φ_1 and Φ_2.

$[\Phi = \Phi_1 \wedge \Phi_2]$ Assume $I \models \Phi$. Then $I \models \Phi_1$ and $I \models \Phi_2$. Also $I \models \text{NEG}(\Phi)$ if $I \models \text{NEG}(\Phi_1)$ or $I \models \text{NEG}(\Phi_2)$, according to rule (ii). By hypothesis, the negation condition holds, so $I \not\models \text{NEG}(\Phi_1)$ and $I \not\models \text{NEG}(\Phi_2)$. Hence, $I \not\models \text{NEG}(\Phi)$.

$[\Phi = \Phi_1 \vee \Phi_2]$ Assume $I \models \Phi$. Then $I \models \Phi_1$ or $I \models \Phi_2$. Also $I \models \text{NEG}(\Phi)$ if $I \models \text{NEG}(\Phi_1)$ and $I \models$

$NEG(\Phi_2)$, according to rule (iii). By hypothesis, the negation condition holds, so $I \not\models NEG(\Phi_X)$ for $X \in \{1, 2\}$. Hence, $I \not\models NEG(\Phi)$.

[$\Phi = (\exists x)(\Phi_1 \wedge \Phi_2)$] Let $NEG(\Phi) = (\forall x)(\Phi_1 \rightarrow \Phi_2)$, according to rule (iv). Rule (v) can be handled similarly. Assume $I \models \Phi$. Then, for some substitution $\sigma = [t/x]$, $I \models \sigma(\Phi_1)$ and $I \models \sigma(\Phi_2)$. By the hypothesis, $I \models \sigma(\Phi_2)$ implies $I \not\models NEG(\sigma(\Phi_2))$. $I \models (\forall x)(\Phi_1 \rightarrow NEG(\Phi_2))$ if $I \models NEG(\sigma(\Phi_2))$ for every $\sigma = [t/x]$ for which $I \models \sigma(\Phi_1)$. We have $I \models \sigma(\Phi_1)$ but $I \not\models NEG(\sigma(\Phi_2))$, so $I \not\models NEG(\Phi)$.

[$\Phi = (\forall x)(\Phi_1 \rightarrow \Phi_2)$] According to rule (vi), $NEG(\Phi) = (\exists x)(\Phi_1 \wedge NEG(\Phi_2))$. Assume $I \models \Phi$. Then $I \models \sigma(\Phi_2)$ whenever $I \models \sigma(\Phi_1)$, for all $\sigma = [t/x]$. By the hypothesis, if $I \models \sigma(\Phi_2)$, then $I \not\models NEG(\sigma(\Phi_2))$. And if $I \models \sigma(\Phi_1)$ and $I \not\models NEG(\sigma(\Phi_2))$, then $I \not\models (\exists x)(\Phi_1 \wedge NEG(\Phi_2))$. Otherwise, $I \not\models \sigma(\Phi_1)$ for all substitutions σ. Again, $I \not\models NEG(\Phi)$. □

For computation purposes, the NEG relation must be compressed into a function *neg*. This is achieved with the following rules for compression of a relation:

- If $NEG(\Phi) = \Phi'$, $NEG(\Phi) = \Phi''$, and $\sigma(\Phi') = \Phi''$ for some substitution σ, then $neg(\Phi) = \Phi'$.
- Also if $NEG(\Phi) = \Phi'$, $NEG(\Phi) = \Phi''$, and Φ' and Φ'' are not unifiable, then $neg(\Phi) = \Phi'$ only. This case requires some method for deciding the appropriate compression.

An extension function can now be defined that builds dual definitions from clauses within a program. Let EXT be an extension function accepting a program Π and a compressed negation function *neg*, returning the smallest program Π' such that:

(i) All clauses of Π are contained in Π'.

(ii) If $L :- \Phi$ is a clause in Π, then $NOT(L) :- neg(\Phi)$ (the dual definition) is in Π'.

The extension $EXT(\Pi, neg)$ preserves the meaning of Π only if Π is *compact*. Program Π is compact if for every clause $L :- \Phi$ in Π there is no other clause $L' :- \Phi'$ such that L and L' are unifiable. In the simple case when a program is not compact due to the presence of clauses $L :- \Phi$ and $L :- \Phi'$, they may be combined to form a clause $L :- \Phi \vee \Phi'$. This reduction does

not alter the least model of the program, as demonstrated in lemma 7.2, below. When the heads of the clauses are not identical, the problem of combining and separating the definitions is more difficult, but solvable.

Proof that clauses L :- Φ and L :- Φ' are equivalent to clause L :- $\Phi \bigvee \Phi'$ is quite simple, and is presented next.

Lemma 7.2: Let $\Pi_1 = \Pi \cup \{L :- \Phi, L :- \Phi'\}$, and $\Pi_2 = \Pi \cup \{L :- \Phi \bigvee \Phi'\}$, where Π is a program. Then $M(\Pi_1) = M(\Pi_2)$.

Proof: The proof is in two parts, each utilizing induction on the structure of the least models, demonstrating first inclusion of $M(\Pi_1)$ in $M(\Pi_2)$ and then inclusion of $M(\Pi_2)$ in $M(\Pi_1)$.

$M(\Pi_1) \subseteq M(\Pi_2)$:

(Basis) *true* $\in M(\Pi_1)$ and *true* $\in M(\Pi_2)$.

(Induction) Assume $M(\Pi_1) \subseteq M(\Pi_2)$. Let σ be a substitution such that $\sigma(L) :- \sigma(\Phi)$ and $\sigma(L) :- \sigma(\Phi')$ are ground instances. If $\Pi_1 \models \sigma(\Phi)$ or $\Pi_1 \models \sigma(\Phi')$, then $\sigma(L) \in M(\Pi_1)$. By the hypothesis, $\Pi_2 \models \sigma(\Phi)$ or $\Pi_2 \models \sigma(\Phi')$, so $\sigma(L) \in M(\Pi_2)$.

$M(\Pi_2) \subseteq M(\Pi_1)$:

(Basis) *true* $\in M(\Pi_2)$ and *true* $\in M(\Pi_1)$.

(Induction) Assume $M(\Pi_2) \subseteq M(\Pi_1)$. If $\sigma(L) :- \sigma(\Phi) \bigvee \sigma(\Phi')$ is a ground clause and $\Pi_2 \models \sigma(\Phi)$ or $\Pi_2 \models \sigma(\Phi')$, then $\sigma(L) \in M(\Pi_2)$. By the hypothesis, $\Pi_1 \models \sigma(\Phi)$ or $\Pi_1 \models \sigma(\Phi')$, so $\sigma(L) \in M(\Pi_1)$. □

I can now demonstrate why programs must be compact. Assume a program Π contains two clauses:

(i) `p :- a.`

(ii) `p :- b.`

Clearly, this program is not compact. By lemma 7.2, these clauses have the same effect as the single clause:

(i′) p :- a ⋁ b.

The extension of Π introduces two new clauses:

~p :- ~a.

~p :- ~b.

Again using lemma 7.2, these new clauses have the same effect as the single clause:

~p :- ~a ⋁ ~b.

Also the extension of clause (i′) introduces the new clause:

~p :- ~a ⋀ ~b.

Clearly the two definitions of ~p do not have the same effect. Thus extension of a non-compact program does not preserve its model-theoretic meaning.

I am now prepared to state the main result of this section:

Lemma 7.3: Π′ = EXT(Π, *neg*) is consistent if:

(i) Π does not contain definitions for *true* and ~*true*;

(ii) The necessary condition for consistency holds for Π;

(iii) Π is compact.

Proof: By induction on subsets M_i of the least model M(Π′).

(Basis) Since *true* ∈ M_0, and ~*true* ∉ M_0, the basis case is consistent.

(Induction) Assume M_i is consistent for $i < n$. Inconsistency can be generated within M_n in the following ways:

(Case 1) There is a ground instance ~*true* :- Φ of a clause in Π′ and $M_i \models \Phi$. But ~*true* :- Φ cannot be an instance of a clause in Π, since this is prohibited by restriction (i). Also EXT(Π, *neg*) does not generate this clause, since Π would then contain a clause of the form *true* :- Φ′, which is also prohibited by restriction (i).

(Case 2) There are ground instances L :- Φ and NOT(L) :- Φ' of clauses in Π', $M_i \models \Phi$ and $M_i \models \Phi'$. By the necessary condition for consistency, these clauses cannot both be instances of clauses in Π. So assume only L :- Φ is an instance of a clause in Π. Because Π is compact, there is no other ground instance of a clause in Π of the form L :- Φ'', where $\Phi'' \neq \Phi$. Hence, it must be that $\Phi' = neg(\Phi)$. By the definition of the negation function *neg*, when M_i is consistent, $M_i \models \Phi$ implies $M_i \not\models \Phi'$. By hypothesis, M_i is consistent, so a contradiction occurs: L $\in M_n$ while NOT(L) $\notin M_n$. □

Hence, when a program Π is compact and satisfies the necessary condition for consistency, EXT(Π, *neg*) forms a conservative extension of Π.

To illustrate the conservative extension of a program, consider the program below:

Example 7.4

(i) `subsequence(nil,M) :- true.`

(ii)
```
subsequence(cons(X,L),cons(Y,M)) :-
        eq(X,Y) ∧ subsequence(L,M).
```

(iii)
```
subsequence(cons(X,L),cons(Y,M)) :-
        subsequence(cons(X,L),M).
```

(iv) `~subsequence(cons(X,L),nil) :- true.`

This program defines the `subsequence` predicate as described for example 4.1. The program in example 7.4 satisfies the necessary condition for consistency. However, the program is not compact, due to the presence of clauses (ii) and (iii). These clauses can be combined to form the following clause:

```
subsequence(cons(X,L),cons(Y,M)) :-
        [eq(X,Y) ∧ subsequence(L,M)]
      ∨ subsequence(cons(X,L),M).
```

Finally, using the rules characterizing the NEG relation presented above, the program presented in example 4.1 results.

8. Summary

This paper serves as an introduction to a logic employing constructive negation. I have demonstrated the soundness and completeness of this logic. I have also argued that constructive negation fits within the procedural semantics of resolution better than negation by failure. Finally, I described the conservative extension of a program. Use of a conservative extension not only prevents formulation of inconsistent programs, it also produces negative definitions from positive definitions. Elaboration of the properties of the conservative extension EXT(Π, *neg*) remains to be done. These properties will rely on a detailed description of the procedural semantics of universal quantification.

Appendix A. Equivalence of SLDCN and SLD Procedures

The syntax chosen for extended Horn-clause programs is slightly expanded from the standard Horn-clause syntax. Standard Horn-clause syntax does not include the following elements:

negative literals,

disjunctive phrases,

existentially quantified phrases.

In all other respects the syntactic forms are identical. Likewise, the SLD procedure is identical to the SLDCN procedure, absent rules for the elements listed above. Extended Horn-clauses will be convenient for describing certain properties of constructive negation. However, it is important to know if a standard Horn-clause interpreter, using SLD-resolution, can successfully execute an extended Horn-clause logic program. This proof is conducted in this section.

A first step in executing extended logic programs with a standard interpreter is to translate extended logic programs into standard logic. This is achieved using a syntactic transformation function TRANS, mapping phrases to phrases. If A is an atom, and Φ_1 and Φ_2 are phrases, the recursive definition of TRANS is as follows:

(i) TRANS[A] = A.

(ii) TRANS[~A] = *not*(A).

(iii) TRANS[$\Phi_1 \wedge \Phi_2$] = TRANS[Φ_1] $\wedge$ TRANS[Φ_2].

(iv) TRANS[$\Phi_1 \vee \Phi_2$] = *or*(TRANS[Φ_1], TRANS[Φ_2]).

(v) TRANS[$(\exists x)\Phi_1$] = *exists*(TRANS[Φ_1]).

TRANS can be extended naturally to map over clauses, and finally over an entire program. Substitution commutes with translation: TRANS[$\sigma(\Phi)$] = σ(TRANS[Φ]).

Execution of translated standard Horn-clause programs by the SLD procedure is achieved by introducing the following clauses, together designated later as Γ:

exists(F) :- F.

or(F1,F2) :- F1.

or(F1,F2) :- F2.

These rules will be used to perform certain higher-order deductions by the SLD-resolution procedure.

Proof of equivalence between SLD and SLDCN procedures relies on their completeness properties. Completeness of SLDCN was demonstrated in the previous section. Completeness of SLD has been shown in [EK76]. Let Π be an extended Horn-clause program with the following restrictions:

(i) Π contains no clauses defining the predicates *not*, *exists*, *or*.

(ii) Every existentially quantified variable has a distinct name.

Then let $\Pi' = \text{TRANS}[\Pi] \cup \Gamma$. The following theorem demonstrates equivalence of least models generated by extended and standard Horn-clause logic programs.

Lemma A.1: $\Pi \models \Phi$ if and only if $\Pi' \models \text{TRANS}[\sigma(\Phi)]$, for all ground phrases Φ, and some substitution σ.

Proof: To show that $\Pi \models \Phi$ implies $\Pi' \models \text{TRANS}[\sigma(\Phi)]$, proceed by induction on the structure of the least model $M(\Pi)$.

(Basis) *true* $\in M(\Pi)$, and *true* $\in M(\Pi')$.

(Induction) Assume the lemma is true for ground phrases Φ_1 and Φ_2 such that $\Pi \models \Phi_1$ and $\Pi \models \Phi_2$. Since the translation function TRANS has no effect for atoms and conjunctions, these cases are trivial, and will not be considered.

[$\Phi = \tilde{}A$] Then TRANS[$\tilde{}A$] = *not*(A). $\Pi \models \tilde{}A$ when there is a clause $\tilde{}A' :- \Phi'$ in Π such that $\rho(A') = A$, $\rho(\Phi') = \Phi_1$, and $\Pi \models \Phi_1$. The clause *not*(A') :- TRANS(Φ') is contained in Π'. By the hypothesis, $\Pi' \models \text{TRANS}[\Phi_1]$, so $\Pi' \models$ *not*(A).

[$\Phi = \Phi_1 \vee \Phi_2$] Then TRANS[$\Phi_1 \vee \Phi_2$] = *or*(TRANS[$\Phi_1$], TRANS[$\Phi_2$]). $\Pi \models \Phi_X$ for $X \in \{1, 2\}$. If

X = 1, there is a clause *or*(F1,F2) :- F1 in Γ, and an instance of this clause is *or*(TRANS[Φ_1, TRANS[Φ_2]) :- TRANS[Φ_1]. By the hypothesis, $\Pi' \models$ TRANS[Φ_1], so $\Pi' \models$ *or*(TRANS[Φ_1, TRANS[Φ_2]). Similarly for X = 2.

[$\Phi = (\exists x)\Phi_1$] Then TRANS[$(\exists x)\Phi_1$] = *exists*(TRANS[Φ_1]). $\Pi \models \Phi$, so $\Pi \models \sigma(\Phi_1)$ for some $\sigma = [t/x]$. There is a clause *exists*(F) :- F in Γ, and an instance of this clause is *exists*(TRANS[$\sigma(\Phi_1)$]) :- TRANS[$\sigma(\Phi_1)$]. By the hypothesis, $\Pi' \models$ TRANS[$\sigma(\Phi_1)$], so $\Pi' \models$ *exists*(TRANS[$\sigma(\Phi_1)$]).

Proof of the converse is achieved in the same manner. □

Theorem A.2: When Π is a program observing the restrictions presented above, Φ is deducible from Π by SLDCN if and only if it is also deducible from program TRANS[Π] ∪ Γ by SLD.

Proof: As a consequence of the completeness theorems for SLD and SLDCN, and use of lemma A.1.

References

[AE82] Apt & van Emden, "Contributions to the Theory of Logic Programming," *JACM*, 29(3), pp. 841-862, 1982.

[C78] Clark, "Negation as Failure", in *Logic and Data Bases*, Gallaire & Minker (eds.), Plenum Press, New York, pp. 55-76, 1978.

[CM81] Clocksin & Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.

[E72] Enderton, *A Mathematical Introduction to Logic*, Academic Press, New York, 1972.

[EK76] van Emden & Kowalski, "The Semantics of Predicate Logic as a Programming Language," *JACM*, 23(4), pp. 733-742, 1976.

[R65] Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *JACM*, 12(1), pp. 23-41, 1965.

[S85] Shepherdson "Negation as Failure II," *Journal of Logic Programming*, 2(3), pp.185-202, 1985.